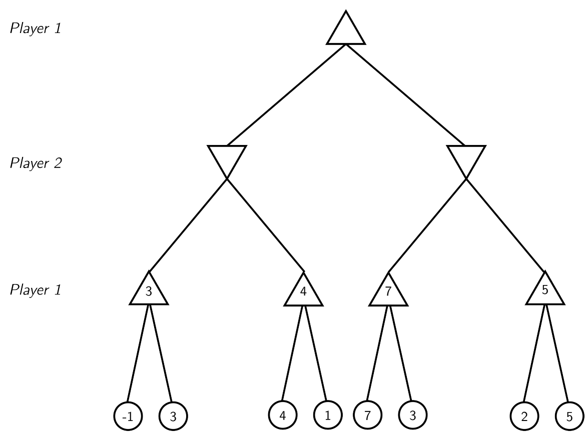
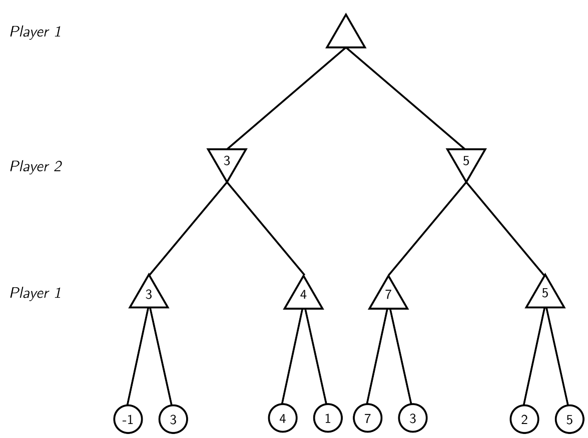
Artificial Intelligence – Lab 04 – Answers

May 10, 2024

Question 1. We can use backward induction, which is equivalent to minimax. Player 1 is playing last, so aims to maximize the value:



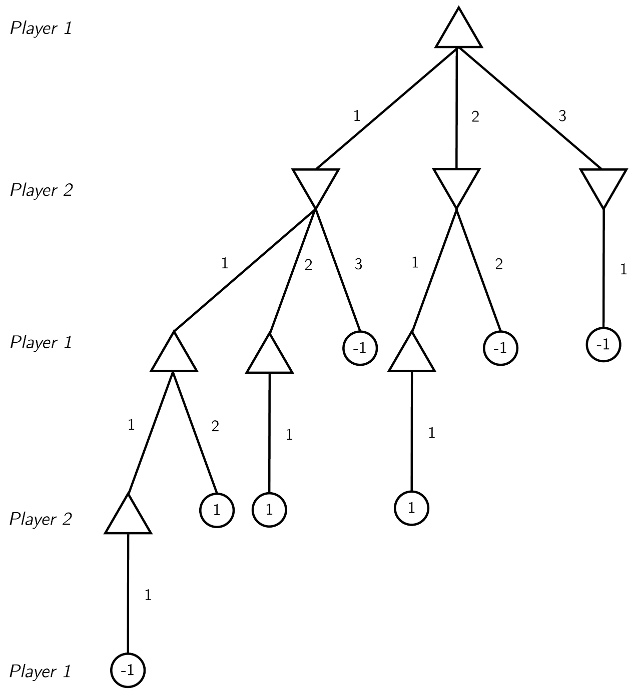
Before that, player 2 was playing, so aiming to minimize the value:



We can now know that, if playing Right first, player 1 is guaranteed to win 5, which is a better payoff than when playing Left.

The optimal sequence is then: [P1: Right, P2: Right, P1: Right]

Question 2. We can represent the task as the following game tree:



where a utility of 1 means that Player 1 (Max) wins and -1 means that Player 2 (Min) wins. Using backward induction, we can verify that a **rational** player 2 will always have a solution that guarantees them to win.

> **Remark:** It doesn't mean that Player 1 would never win against any possible Player 2. If Player 2 doesn't play rationally, it may offer the possibility to Player 1 to win. Minimax algorithm is guaranteed to be optimal only when the two agents are optimal (i.e. fully rational).

Question 3.

```python
def minimax_search(state: NimState) -> NimAction:
    max_utility = -math.inf
    argmax = None
    for action in available_actions(state):
        v = min_value(result(state, action))
        if v > max_utility:
            max_utility = v
            argmax = action
    return argmax


def max_value(state: NimState) -> int:
    if is_goal(state):
        return utility(state)
    v = -math.inf
    for action in available_actions(state):
        v = max(v, min_value(result(state, action)))
    return v
```

```python
def min_value(state: NimState) -> int:
    if is_goal(state):
        return utility(state)
    v = math.inf
    for action in available_actions(state):
        v = min(v, max_value(result(state, action)))
    return v
```

Question 4.

```python
def maximin_search(state: NimState) -> NimAction:
    min_utility = math.inf
    argmin = None
    for action in available_actions(state):
        v = max_value(result(state, action))
        if v < min_utility:
            min_utility = v
            argmin = action
    return argmin
```

One can observe that Min wins whenever N is a multiple of 4 and that Max wins otherwise.

Question 5.

a) Let $N = 4k$ for some $k \in \mathbb{N}$. We proceed by induction over k. The base case was proved in Question 2. Suppose Max removes $m \in \{1, 2, 3\}$ objects from the pile. A rational Min can then remove $4 - m$ objects, causing Max to be faced with $4(k - 1)$ objects. By the induction hypothesis, Max loses this game against a rational Min.

b) If N is not a multiple of 4, it means that $N = 4k + \ell$ for some $k \in \mathbb{N}$ and some $\ell \in \{1, 2, 3\}$. Max can thus remove ℓ objects from the pile, causing Min to be faced with $4k$ objects. By a), with the roles of Min and Max swapped, we know that Min must lose against a rational opponent.

> **Remark:** This argument can also be generalized to variants of the game where players are allowed to remove up to $m \in \mathbb{N}$ objects at a time.

Question 6.

```python
def alpha_beta_search(state: TicTacToeState) -> TicTacToeAction:
    global count
    count = 0

    value, action = max_value(state, -math.inf, math.inf)
    return action


def max_value(
    state: TicTacToeState, alpha: int, beta: int
) -> tuple[int, TicTacToeAction]:
    global count
    count += 1

    if is_goal(state):
```

```python
        return utility(state), None
    v = -math.inf
    best_action = None

    for action in available_actions(state):
        v2, _ = min_value(result(state, action), alpha, beta)
        if v2 > v:
            v, best_action = v2, action
            alpha = max(alpha, v)
        if v >= beta:
            break
    return v, best_action


def min_value(
    state: TicTacToeState, alpha: int, beta: int
) -> tuple[int, TicTacToeAction]:
    global count
    count += 1

    if is_goal(state):
        return utility(state), None
    v = math.inf
    best_action = None
    for action in available_actions(state):
        v2, _ = max_value(result(state, action), alpha, beta)
        if v2 < v:
            v, best_action = v2, action
            beta = min(beta, v)
        if v <= alpha:
            break
    return v, best_action
```

Because the AI plays rationally, you can never win against it, but you can end the game in a draw.

Question 7. The number of recursive calls changes, as the order in which actions are explored is shuffled. This is because of the way alpha-beta-search works. Here is an example: Let y and z be two MIN-nodes, which are children of the MAX-node x. Suppose the values of y and z are 1 and 2 respectively (computed by doing minimax on their children). If y is considered before z, all computations will be performed as usual. However, if z is considered before y, we will have $\alpha(x) = 2$, so that, as soon as one of y's children yields the value 1, the computation will stop: No matter what values the remaining children yield, y's value will be *at most* 1, and therefore completely irrelevant for x, as it already "knows" it can get a higher value of 2 from z.